PRÉCISION DE 100 DÉCIMALES EN C++ ET PYTHON

PETER PHILIPPE

Par défaut, les langages de programmation usuels permettant d'effectuer des calculs comme C, C++, Python et bien d'autres comme notamment Ada, C#, Fortran, Java, JS, Julia, Nim, OCaml, Rust, Swift, Scala, Zig, etc. proposent une précision maximale de 64 bits (soit ≈ 15 décimales), voir éventuellement de 128 bits (≈ 32 décimales). Ce petit article présente deux librairies bien connues permettant d'étendre considérablement la précision d'un résultat pour les nombres flottants et ce, quelque soit le système d'exploitation. Cette précision sera volontairement limitée à cent décimales, mais il sera assez facile de modifier les codes sources pour l'accroître, afin d'aller bien au-delà de cette limite arbitrairement choisie. Comme d'habitude, les codes sources et ce fichier PDF seront disponibles à cette adresse `https://github.com/rayptor/linkedin`.

Les langages **C** et **C++** disposent publiquement depuis février 2000 de la librairie nommée **MPFR**[1] développée originellement à l'INRIA par Paul Zimmermann. Un *wrapper* de cette librairie a été intégré dans la librarie Boost[2] pour le module `multiprecision` développé initialement par Christopher Kormanyos en 2002, c'est de cette version dont il sera question ici.

En ce qui concerne **Python**, il existe entre autres le module Decimal[3] intégré en standard, la librairie Gmpy2[4] ainsi que la librairie **Mpmath**[5], c'est cette dernière qui nous intéressera pour illustrer ce sujet. Elle a été développée en 2007 par Fredrik Johansson et permet non seulement d'avoir accès au calcul en haute précision sur des nombres réels et complexes, mais également pour différentes classes d'algorithmes numériques (recherche des racines, systèmes linéaires et éléments propres, intégration et dérivation, théorie des nombres, etc.).

L'une des branches du calcul scientifique se prêtant particulièrement bien au sujet de cette note, est celle de la recherche des zéros d'une fonction. Prenons comme méthode celle de Geum Y. H. et Kim Y. I. permettant la recherche d'une racine simple α pour une équation non linéaire univariée f telle que $f(\alpha) = 0$, elle est intitulée *A biparametric family of optimally convergent sixteenth-order multipoint methods with their fourth-step weighting function as a sum of a rational and a generic two-variable*

1. https ://www.mpfr.org
2. https ://www.boost.org
3. https ://docs.python.org/3/library/decimal.html
4. https ://github.com/aleaxit/gmpy
5. https ://mpmath.org

function. Cet article scientifique est accessible gratuitement sur Science Direct [6]. La présentation de cette méthode itérative multipoints à quatre pas, dont l'ordre de convergence est de 16, ne sera pas détaillée, car ce n'est pas la finalité recherchée.

Passons alors directement au schéma itératif qui sera programmé, par rapport à la publication originale, il correspond à la formule 1.7, les fonctions de pondérations $\mathcal{K}_f$, $\mathcal{H}_f$ et $\mathcal{W}_f$ sont celles de la formule 1.8 et la fonction analytique $G(u,w)$ correspond à la méthode **Y1** de la table 1 où les valeurs de β et de σ valent respectivement 2 et -2 :

$$\boxed{\begin{aligned}
y_n &= x_n - \frac{f(x_n)}{f'(x_n)}\\
z_n &= y_n - \mathcal{K}_f(u_n)\frac{f(y_n)}{f'(x_n)}\\
s_n &= z_n - \mathcal{H}_f(u_n,v_n,w_n)\frac{f(y_n)}{f'(x_n)}\\
x_{n+1} &= z_n - \mathcal{W}_f(u_n,v_n,w_n,t_n)\frac{f(s_n)}{f'(x_n)}\\
&\text{avec}\\
u_n &= \frac{f(y_n)}{f(x_n)}\\
v_n &= \frac{f(z_n)}{f(y_n)}\\
w_n &= \frac{f(z_n)}{f(x_n)}\\
t_n &= \frac{f(s_n)}{f(z_n)}\\
&\text{et}\\
\mathcal{K}_f &= \frac{1+\beta u_n + \left(-9+\dfrac{5\beta}{2}\right)u_n^2}{1+(\beta-2)u_n+(-4+2\beta)u_n^2}\\
\mathcal{H}_f &= \frac{1+2u_n+(2+\sigma)w_n}{1-v_n+\sigma w_n}\\
\mathcal{W}_f &= \frac{1+2u_n+(2+\sigma)v_n w_n}{1-v_n-2w_n-t_n+2(1+\sigma)v_n w_n} + G(u_n,w_n)\\
\phi_1 &= 11\beta^2 2 - 66\beta + 136\\
\phi_2 &= 2u_n(\sigma^2-2\sigma-9) - 4\sigma - 6\\
G(u_n,w_n) &= -\frac{1}{2}(u_n w_n(6+12u_n+u_n^2+u_n^2(24-11\beta)+u_n^3\phi_1+4\sigma)) + \phi_2 w_n^2
\end{aligned}}$$

6. https ://www.sciencedirect.com/science/article/pii/S0377042711000045

Ce modèle sophistiqué est nettement plus performant que celui de Newton-Raphson pour lequel j'ai eu fait un article il y a peu.

L'équation de test pour les deux codes sources est de degré impair, ce qui offre la garantie d'avoir au moins une racine dans $\mathbb{R}$:

$$f(x) = 13x^9 - x\exp(x^7) + x^2 + \cos(x) + \frac{\sqrt{8}}{x^3}$$

Dont la dérivée est :

$$f'(x) = 117x^8 - \exp(x^7)(7x^7 + 1) + 2x - \sin(x) - \frac{6\sqrt{2}}{x^4}$$

Voici tout d'abord le fichier Makefile permettant de compiler le fichier source en C++ :

```
# Fichier Makefile
CXX := g++-14
CXXFLAGS := -march=native -flto=auto -O3 -std=gnu++23 -Wall
INCFLAGS := -I/usr/local/include/
LDFLAGS := -L/usr/local/lib/
LDLIBS := -lmpfr

TARGET := geum_kim_boost
SOURCE := geum_kim_boost.cpp

all: $(TARGET)

$(TARGET): $(SOURCE)
  $(CXX) $(CXXFLAGS) $(INCFLAGS) $(LDFLAGS) $(LDLIBS) $(SOURCE) -o $(TARGET)

clean:
  rm -f $(TARGET)
```

Et le code en C++ avec la libraire Boost, bien entendu il est loin d'être parfait et pourra faire l'objet de plusieurs améliorations, l'objectif étant simplement de faire une modeste démonstration. Pour des raisons de mise en page, plusieurs lignes de ce source ont dû être scindées en deux lignes :

```
// fichier geum_kim_boost.cpp

#include <iostream>
#include <limits>
#include <format>
#include <iomanip>
#include <functional>
#include <cstdlib>
#include <string>
#include <stdexcept>
#include <boost/multiprecision/mpfr.hpp>

using namespace boost::multiprecision;
using t_mpfr = mpfr_float_100;

template <typename T> T geum_kim(
       std::function<T(T)> f,
       std::function<T(T)> df,
       T x0,
       std::size_t maxit) {

    T xOld = x0;
    T xNew = x0;
    std::size_t k = 0;
    const T tol = std::numeric_limits<T>::epsilon();
    const T beta = static_cast<T>(2);
    const T sigma = -beta;
    const T sigmaSqr = sigma * sigma;
```

```
const T phi1 = static_cast<T>(11) * beta * beta - static_cast<T>(66) * beta
       + static_cast<T>(136);

while (k < maxit) {
    T fxn = f(xNew);
    T dfxn = df(xNew);

    if (abs(dfxn) < std::numeric_limits<T>::min()) {
        std::cerr << "Dérivée de f(x) trop petite àl'itération "
               << k << "risque de division par zéro." << std::endl;
        break;
    }

    T yn = xOld - fxn / dfxn;
    T fyn = f(yn);
    T un = (abs(fxn) > tol) ? (fyn / fxn) : static_cast<T>(0);
    T un2 = un * un;
    T phi2 = static_cast<T>(2) * un * (sigmaSqr - static_cast<T>(2) * sigma - static_cast<T>(9))
           - static_cast<T>(4) * sigma - static_cast<T>(6);

    T kfNum = (static_cast<T>(1) + beta * un + (static_cast<T>(-9) + (static_cast<T>(5) * beta)
    / static_cast<T>(2)) * un2);
    T kfDen = (static_cast<T>(1) + (beta - static_cast<T>(2)) * un + (static_cast<T>(-4) + beta
    / static_cast<T>(2)) * un2);
    if (abs(kfDen) < std::numeric_limits<T>::min())
        std::cerr << "Le dénominator kfDen est trop petit àl'itération "
        << k << "risque de division par zéro." << std::endl;
    T kf = kfNum / kfDen;

    T zn = yn - kf * (fyn / dfxn);
    T fzn = f(zn);
    T vn = (abs(fyn) > tol) ? (fzn / fyn) : static_cast<T>(0);
    T wn = (abs(fxn) > tol) ? (fzn / fxn) : static_cast<T>(0);

    T hfNum = (static_cast<T>(1) + static_cast<T>(2) * un + (static_cast<T>(2) + sigma) * wn);
    T hfDen = (static_cast<T>(1) - vn + sigma * wn);
    if (abs(hfDen) < std::numeric_limits<T>::min())
        std::cerr << "Dénominator hfDen trop petit àl'itération "
        << k << "risque de division par zéro." << std::endl;
    T hf = hfNum / hfDen;

    T sn = zn - hf * (fzn / dfxn);
    T fsn = f(sn);
    T tn = (abs(fzn) > tol) ? (fsn / fzn) : static_cast<T>(0);

    T guw1 = static_cast<T>(6) + static_cast<T>(12) * un + static_cast<T>(2) * un2;
    T guw2 = (static_cast<T>(24) - static_cast<T>(11) * beta) + pow(un, 3U) * phi1
           + static_cast<T>(4) * sigma;
    T guw = (static_cast<T>(-0.5)) * un * wn * (guw1 * guw2) + phi2 * wn * wn;

    T wfNum = (static_cast<T>(1) + static_cast<T>(2) * un + (static_cast<T>(2) + sigma) * vn * wn);
    T wfDen = (static_cast<T>(1) - vn - static_cast<T>(2) * wn - tn + static_cast<T>(2)
           * (static_cast<T>(1) + sigma) * vn * wn) + guw;
    if (abs(wfDen) < std::numeric_limits<T>::min())
        std::cerr << "Dénominator wfDen trop petit àl'itération "
            << k << "risque de division par zéro." << std::endl;
    T wf = wfNum / wfDen;

    xNew = sn - wf * (fsn / dfxn);

    T delta = abs(xNew - xOld);
    if (delta < tol) {
        std::cerr << "Convergence terminée :" << std::endl;
        break;
    }

    if constexpr (std::numeric_limits<T>::is_specialized && std::numeric_limits<T>::digits10 > 0) {
        std::cout << std::fixed << std::setprecision(std::numeric_limits<T>::digits10)
            << std::format("Itération numéro{:3d} -> X = {}\n", k,
                xNew.str(std::numeric_limits<T>::digits10, std::ios_base::fixed));
    } else {
        std::cout << std::fixed << std::setprecision(100)
            << std::format("Itération numéro{:3d} -> X = {}\n", k,
```

```
                xNew.str(100, std::ios_base::fixed));
        }

        xOld = xNew;
        ++k;
    }
    if (k == maxit) {
        std::cerr << "Nombre maximal d'itérations " << maxit << " atteint !" << std::endl;
    }

    return xNew;
}

int main(int argc, char* argv[]) {
    t_mpfr valeurInitiale = t_mpfr("1.5");
    std::size_t iterationsMax = 20;
    int precisionSortie = 0;
    std::string msgDef = "Utilisation de la valeur par défaut : ";
    mpfr_float::default_precision(100);

    if (argc > 1) {
        try {
            valeurInitiale = t_mpfr(argv[1]);
        } catch (const std::exception& e) {
            std::cerr << "Valeur pour x0 invalide (" << argv[1] << ") !"
                    << msgDef << " '1.5'. Erreur : " << e.what() << "\n";
            valeurInitiale = t_mpfr("1.5");
        }
    }
    if (argc > 2) {
        try {
            long iterationsMaxUser = std::stol(argv[2]);
            if (iterationsMaxUser <= 0) {
                std::cerr << "Nombre maximum d'itérations négatif (" << argv[2] << ")! Doit être > 0. "
                        << msgDef << iterationsMax << std::endl;
                 iterationsMax = 20;
            } else {
                iterationsMax = static_cast<std::size_t>(iterationsMaxUser);
            }
        } catch (const std::invalid_argument& invArg) {
            std::cerr << "Nombre d'itérations invalide (" << argv[2] << ") ! "
                    << msgDef << iterationsMax << "Erreur : " << invArg.what() << std::endl;
            iterationsMax = 20;
        } catch (const std::out_of_range& outOr) {
            std::cerr << "Nombre d'itérations hors limites (" << argv[2] << ") ! "
                    << msgDef << iterationsMax << "Erreur : " << outOr.what() << std::endl;
            iterationsMax = 20;
        }
    }
    std::cout << "Valeur initiale de x0 = " << valeurInitiale << " avec " << iterationsMax
            << " itérations." << std::endl;

    auto f = [&](t_mpfr x) -> t_mpfr {
        return static_cast<t_mpfr>(13) * pow(x, 9U) - x * exp(pow(x, 7U)) + cos(x) + sqrt(static_cast<t_mpfr>(8))
                / pow(x, 3U) + pow(x, 2U);
    };

    auto df = [&](t_mpfr x) -> t_mpfr {
        return static_cast<t_mpfr>(117) * pow(x, 8U) - exp(pow(x, 7U)) * (static_cast<t_mpfr>(7) * pow(x, 7U)
                + static_cast<t_mpfr>(1)) - sin(x) - (static_cast<t_mpfr>(6) * sqrt(static_cast<t_mpfr>(2)))
                / pow(x, 4U) + static_cast<t_mpfr>(2) * x;
    };

    if constexpr (std::numeric_limits<t_mpfr>::is_specialized && std::numeric_limits<t_mpfr>::max_digits10 > 0) {
        precisionSortie = std::numeric_limits<t_mpfr>::max_digits10;
    }

    t_mpfr resultat = geum_kim<t_mpfr>(f, df, valeurInitiale, iterationsMax);
    std::cout << std::fixed << std::setprecision(precisionSortie)
            << std::format("\n\t -> X = {}\n", resultat.str(precisionSortie, std::ios_base::fixed)) << std::endl;

    return 0;
}
```

Et maintenant la version pour le langage Python 3 avec la librairie MPMATH, ce code bien que calqué sur la version en C++, est également perfectible, mais là aussi seul l'aspect purement fonctionnel est retenu pour l'exemple :

```
# fichier geum_kim_mpmath.py

from mpmath import mp, mpf, sin, cos, sqrt, exp, power, fabs
from typing import Callable, Any

mp.dps = 100

def geum_kim(
        f: Callable[[Any], Any],
        df: Callable[[Any], Any],
        x0: Any,
        maxit: int,
    ) -> Any:

    xOld = mpf(x0)
    xNew = mpf(xOld)
    k = 0
    tol = power(mpf(10), -(mp.dps - 5))
    beta = mpf(2)
    sigma = -beta
    sigmaSqr = power(sigma, 2)
    phi1 = mpf(11) * power(beta,2) - mpf(66) * beta + mpf(136)

    while k < maxit:
        fxn = f(xNew)
        dfxn = df(xNew)

        if fabs(dfxn) < mp.eps:
            print(f"Dérivée de f(x) trop petite àl'itération {k}.")
            break

        yn = xOld - fxn / dfxn
        fyn = f(yn)
        un = mpf(0) if fabs(fxn) < tol else fyn / fxn
        un2 = power(un, 2)
        phi2 = mpf(2) * un * (sigmaSqr - mpf(2) * sigma - mpf(9)) - mpf(4) * sigma - mpf(6)

        kfNum = (mpf(1) + beta * un + (mpf(-9) + (mpf(5) * beta) / mpf(2)) * un2)
        kfDen = (mpf(1) + (beta - mpf(2)) * un + (mpf(-4) + beta / mpf(2)) * un2)
        if fabs(kfDen) < mp.eps:
            print(f"Le dénominator kfDen est trop petit àl'itération {k}.")
        kf = kfNum / kfDen

        zn = yn - kf * (fyn / dfxn)
        fzn = f(zn)
        vn = mpf(0) if fabs(fyn) < tol else fzn / fyn
        wn = mpf(0) if fabs(fxn) < tol else fzn / fxn

        hfNum = (mpf(1) + mpf(2) * un + (mpf(2) + sigma) * wn)
        hfDen = (mpf(1) - vn + sigma * wn)
        if fabs(hfDen) < mp.eps:
            print(f"Dénominator hfDen trop petit àl'itération {k}.")
            break
        hf = hfNum / hfDen

        sn = zn - hf * (fzn / dfxn)
        fsn = f(sn)
        tn = mpf(0) if fabs(fzn) < tol else fsn / fzn

        guw1 = mpf(6) + mpf(12) * un + mpf(2) * un2
        guw2 = (mpf(24) - mpf(11) * beta) + power(un,3) * phi1 + mpf(4) * sigma
        guw = (mpf(-0.5)) * (un * wn * (guw1 * guw2)) + phi2 * power(wn,2)

        wfNum = (mpf(1) + mpf(2) * un + (mpf(2) + sigma) * vn * wn)
        wfDen = (mpf(1) - vn - mpf(2) * wn - tn + mpf(2) * (mpf(1) + sigma) * vn * wn) + guw
        if fabs(wfDen) < mp.eps:
            print(f"Dénominator wfDen trop petit àl'itération {k} risque de division par zéro.")
            break
```

```
        wf = wfNum / wfDen

        xNew = sn - wf * (fsn / dfxn)

        delta = fabs(xNew - xOld)
        if delta < tol:
            print(f"Convergence terminée :")
            break

        print(f"Itération numéro{k:3d} -> X = {xNew}")
        xOld = xNew
        k += 1
    else:
        print(f"Nombre maximal d'itérations '{maxit}' atteint !")

    return xNew


if __name__ == "__main__":
    valeurInitiale = mpf("1.5")
    iterationsMax = int(20)
    iterationsMaxUser = int(0)
    msgDef = "Utilisation de la valeur par défaut : "

    if len(sys.argv) > 1:
        try:
            valeurInitiale = mpf(sys.argv[1])
        except Exception as e:
            print(f"Valeur pour x0 invalide ({sys.argv[1]}) ! {msgDef} '1.5'. Erreur : {e}", file=sys.stderr)
            valeurInitiale = mpf("1.5")

    if len(sys.argv) > 2:
        try:
            iterationsMaxUser = int(sys.argv[2])
            if iterationsMaxUser <= 0:
                print(f"Nombre maximum d'itérations négatif ({sys.argv[2]}) ! Doit être > 0. \
                      {msgDef}{iterationsMax}", file=sys.stderr)
                iterationsMax = 20
            else:
                iterationsMax = iterationsMaxUser
        except ValueError as invArg:
            print(f"Nombre d'itérations invalide ({sys.argv[2]}) ! {msgDef}{iterationsMax} \
                  Erreur : {invArg}", file=sys.stderr)
            iterationsMax = 20
        except OverflowError as outOr:
            print(f"Nombre d'itérations hors limites ({sys.argv[2]}) ! {msgDef}{iterationsMax} \
                  Erreur : {outOr}", file=sys.stderr)
            iterationsMax = 20

    print(f"Valeur initiale de x0 = {valeurInitiale} avec {iterationsMax} itérations.")

    f = lambda x: mpf(13)*power(x,9) - x * exp(power(x,7)) + cos(x) + sqrt(mpf(8)) / power(x,3) + power(x,2)
    df = lambda x: mpf(117)*power(x,8) - exp(power(x,7))*(mpf(7)*power(x,7) + mpf(1)) - sin(x) - \
        (mpf(6)*sqrt(mpf(2))) / power(x,4) + mpf(2)*x

    resultat = geum_kim(f, df, valeurInitiale, iterationsMax)
    print(f"\n\t -> X = {resultat}\n")
```

La page suivante montre l'exécution des deux programmes avec la même valeur initiale $x_0 = 1{,}6$ et un nombre d'itérations maximal commun fixé à 10 :

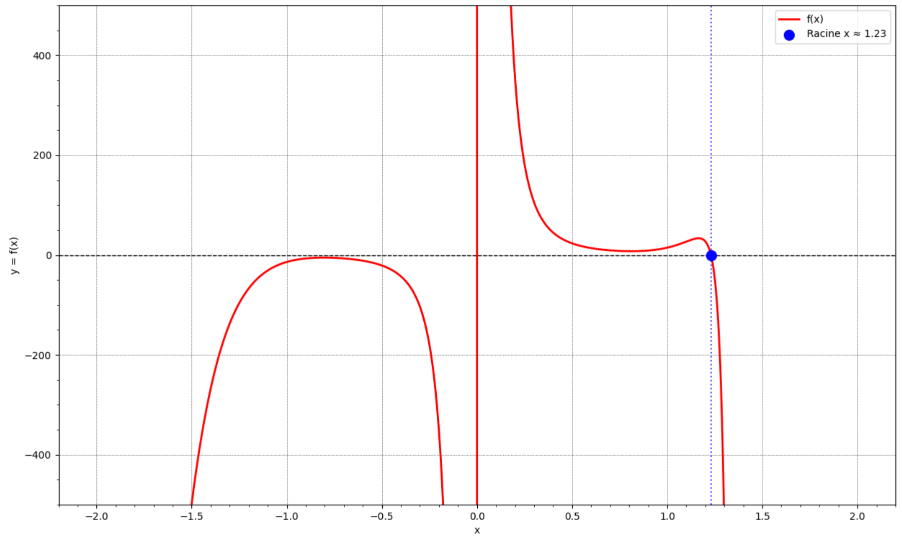
```
% ./geum_kim_boost 1.6 10
Valeur initiale de x0 = 1.6 avec 10 itérations.
Itération numéro 0 -> X = 1.5679765816259249586137627999119078605181449138391004762395961829690455593171970862822474715555873698
Itération numéro 1 -> X = 1.5315043210183152169239018935022548199862420084575412553919430630176255884175656357169106393688556262
Itération numéro 2 -> X = 1.4889874949557683931794198032291807075702962526242667615962023732520349867915815641638097963863039156
Itération numéro 3 -> X = 1.4377453130885255692100120176741793288226295123450921913755453452871665820744550304327050626443856740
Itération numéro 4 -> X = 1.3728976431655516591156613867048642596106418716167244480222703410224120192432770030169730422020133857
Itération numéro 5 -> X = 1.2891014015621215840680200853321531944573077989593515463824248531862568329680266221599177630834665292
Itération numéro 6 -> X = 1.2316520459719357955534572950565560619926041545168948057218993281090237432575443102971888867049891560
Itération numéro 7 -> X = 1.2300661769313939982570274128692085267984247325874529135133917874333720187448907449440505625930693491
Itération numéro 8 -> X = 1.2300661769313939883048363575105749327420065385728738279502693565376326420139071961923314181965381677
Convergence terminée :

        -> X = 1.230066176931393988304836357510574932742006538572873827950269356537632642013907196192331418196538167669

% python3 geum_kim_mpmath.py 1.6 10
Valeur initiale de x0 = 1.6 avec 10 itérations.
Itération numéro 0 -> X = 1.567976581625924958613762799911907860518144913839100476239596182969045559317197086282247471555587369
Itération numéro 1 -> X = 1.531504321018315216923901893502254819986242008457541255391943063017625588417565635716910639368855626
Itération numéro 2 -> X = 1.488987494955768393179419803229180707570296252624266761596202373252034986791581564163809796386303915
Itération numéro 3 -> X = 1.437745313088525569210012017674179328822629512345092191375545345287166582074455030432705062644385674
Itération numéro 4 -> X = 1.372897643165551659115661386704864259610641871616724448022270341022412019243277003016973042202013386
Itération numéro 5 -> X = 1.28910140156212158406802008533215319445730779895935154638242485318625683296802662215991776308346653
Itération numéro 6 -> X = 1.231652045971935795553457295056556061992604154516894805721899328109023743257544310297188886704989156
Itération numéro 7 -> X = 1.230066176931393998257027412869208526798424732587452913513391787433372018744890744944050562593069349
Itération numéro 8 -> X = 1.230066176931393988304836357510574932742006538572873827950269356537632642013907196192331418196538168
Convergence terminée :

        -> X = 1.230066176931393988304836357510574932742006538572873827950269356537632642013907196192331418196538168
```

La racine recherchée $x^* \simeq 1{,}23\ldots$ est bien celle indiquée dans ce premier graphique dans $[-2{,}2]$:



Et ici avec un petit agrandissement dans $[1{,}1.4]$ permettant de distinguer avec davantage de concision la valeur de la racine précédemment calculée :